# Trollius Library Reference in Fortran

Trollius is a programming environment for message passing MIMD parallel computers (multicomputers). This document covers all aspects of Trollius Fortran programming for application programmers. It is not a complete reference for system programmers. New users are referred to the *Trollius Tutorial Introduction in Fortran*. *Trollius Command Reference* describes how to operate a Trollius multicomputer. A version of this document is available for C programmers.

Ohio Supercomputer Center

Research Computing
The Ohio State University

**Document Series 2/4** **March 29, 1992**

## How to Use This Document

This document is a Trollius programmer's reference describing all user level functions. Extra emphasis is placed on the critical aspects of multicomputer parallel programming: message passing, process management and file access.

This document is user oriented. It does not give much insight into how the system is implemented. It does not detail every option and capability of every function. The manual pages cover all the functions in great detail and are meant to supplement this document.

We have kept the font and syntax conventions to a minimum.

| | |
|---|---|
| `code` | This font is used for things you type on the keyboard or see printed on the screen. We use it in code sections and tables but not in the main text. |
| *Section* | Italics are used to cross reference another section in the document or another document. |

## Table of Contents

## Introduction

Trollius presents a broad view of the multicomputer architecture. It recognizes that any computer reachable via communication link(s) is a potential participant in a message passing application. The software presents a uniform programming environment across general purpose computers (e.g.: workstations) and dedicated highly integrated computer ensembles. The latter is usually singled out and tagged as the "parallel machine" and is packaged into an impressive looking box. In Trollius, we refer to these computers as Inside The Box computers or ITB. We recognize the following characteristics of ITB computers:

- They are dedicated to single users.
- All incoming and outgoing links are dedicated to single users.
- They are great in number.
- They are going to be used for large parallel computational problems and not the common applications that UNIX workstations do well.
- On average, they have few peripherals.
- They will run Trollius natively.

Since Trollius is not going to provide all the services of a general purpose operating system, the ITB computers must be hosted by a computer with such an operating system. The host computer will have one or more communication links to one or more ITB computers. As indicated above, other general purpose computers may participate in a multicomputer if the communication links exist such that all computers of both types are reachable (directly or indirectly by hopping through intermediate computers). In most cases, general purpose computers, be they workstations or mainframes, are reachable via a Local Area Network (LAN). In Trollius, we refer to general purpose computers as Outside The Box computers or OTB. We recognize the following characteristics of OTB computers:

- They are shared among multiple users.
- Links to other OTB computers are shared among multiple users.
- They may control peripherals needed by a parallel application.
- They will run Trollius as an application of the native, general purpose operating system.

When running Trollius, we refer to ITB or OTB computers as nodes. Very often we make no node type distinction when describing functionality - because there is none. Unless the node type is specified, assume the function applies generally to any node.

Figure 1 represents the broad Trollius multicomputer.

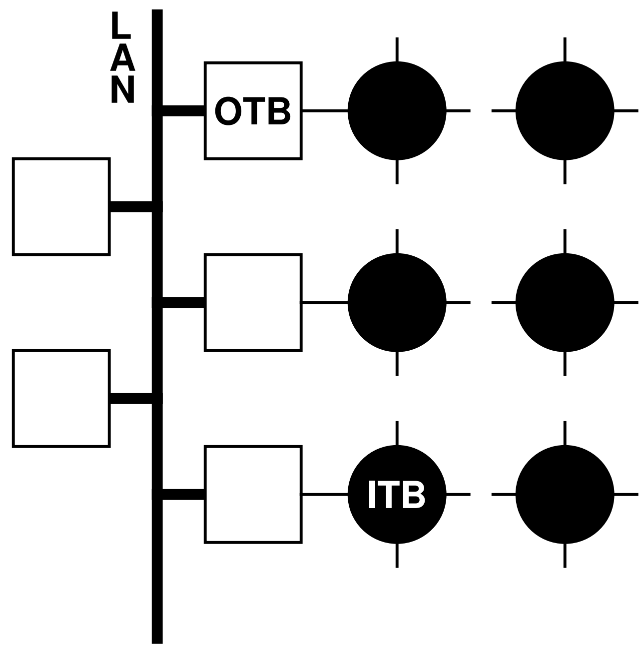


**Figure 1: Trollius Multicomputer Architecture**

**Basic Objects**

Trollius does not restrict the interconnection topology of nodes. A globally unique 32 bit identifier is given to each node (nodeid), chosen by the user. Many Trollius functions take the nodeid as their first argument. The node is the only global object in Trollius. The two other important objects are purely local to each node. They are the process and the event.

Every Trollius node has a multi-tasking, message passing local environment upon which the rest of the operating system is built. There is a small kernel at each node that manages and synchronizes any number of local processes. Each process has a locally unique 32 bit identifier (pid). Many Trollius functions, though not message passing, operate on pids.

The event is an arbitrary 32 bit quantity and the key synchronization point for two local processes. You will see an event in three places.

- in the state of a process blocked on sending a message
- in the state of a process blocked on receiving a message
- in a network (remote node) message

Equal events synchronize. The event has an associate, subordinate object called the type which is found wherever the event is found. It is the secondary synchronization point between two local processes. It is irrelevant if the event does not synchronize. It is irrelevant if it is set to zero. Non-zero types that share a common one bit (bitwise AND is non-zero) synchronize.

**Fortran Programs under Trollius**

A Fortran program that will run under Trollius is the same as a serial Fortran program in terms of variable types, statements, control flow, and functions. Program structure is virtually the same. A Fortran program running under Trollius is simply a Fortran program that makes use of the special Trollius functions for message passing, process creation, etc.

Writing an individual Trollius program is thus very similar to writing a serial program. An extra level of complexity is added, naturally, in designing processes that will run in parallel. Programmers must deal with decomposition, data alignment, synchronization and load balancing. These topics are beyond the scope of this reference. We can state, however, that Trollius can be used to implement many different models of parallel programming from data domain decomposition to neural networks.

**Return Values and Error Reporting**

Unless stated otherwise, Trollius subroutines return 0 in a value returned argument upon successful execution and a non-zero code if an error was encountered. Trollius error codes are listed in *Appendix B: Error Codes*.

| | |
|---|---|
| **`KINIT()`** | Initialize Trollius process. |
| **`KEXIT()`** | Terminate Trollius process. |

## Initialization and Termination

Before Trollius functions may be called, a process must initialize Trollius related data structures and attach to the Trollius kernel. Both of these functions are accomplished by KINIT(). The first argument is the priority, which will be discussed in detail later in this section. In most cases, the priority should be set to 0.

```
subroutine KINIT(priority, ierr)
integer priority, ierr
subroutine KEXIT(status)
integer status
```

ITB programs call KINIT() with priority 0 prior to entering the mainline code. If priority 0 is satisfactory, ITB programs need not call KINIT(), though it is recommended that they do, for portability. KINIT() registers with the local kernel. After KINIT() other kernel requests may be made. KEXIT() says good-bye to the kernel and terminates the program. The argument passed to KEXIT() is an error code with 0 indicating that no error occurred. ITB programs will call KEXIT(0) after a stop statement.

### Controlled and Uncontrolled Scheduling

Kernel scheduling is a confusing topic that is best avoided by always attaching processes at priority 0. This puts the process into a category called uncontrolled. Scheduling uncontrolled processes is very simple. When a process is not blocked on a message, it receives an immediate reply from the kernel and then it is up to the underlying environment[1] to schedule the process. We have no control over the underlying scheduling policy and sometimes it is inadequate for the basic process structuring needs of an application; it may be inadequate for some Trollius system processes.

We occasionally need something more like real-time scheduling: a multipriority environment where the highest priority unblocked process runs until it blocks (with no time-slicing). The Trollius kernel provides such

1. The underlying environment is a lower level kernel providing some form of multi-process scheduling and interprocess communication. Transputers provide this in hardware. Other ITB microprocessors run a software sub-kernel. On Trollius OTB nodes, UNIX provides the scheduling and IPC. It is important to understand that the Trollius kernel is a server and not an executive. It is scheduled along with its clients by the underlying environment.

scheduling to processes with non-zero priorities, called controlled processes. Only the highest priority unblocked controlled process receives a reply from the kernel and is allowed to be scheduled by the underlying environment. There may be other, lower priority unblocked processes (which would be released if they were uncontrolled) but they are not allowed to run. Eventually, the active controlled process will block and the kernel will release the next highest priority unblocked process. A caveat for real-time applications is that there is currently no pre-emption capability.

Unfortunately, there is an opportunity for further complication. When a controlled process synchronizes with an uncontrolled process, we have learned through experience that it is highly desirable to impede the uncontrolled process until the controlled process is released and then blocks. In other words, the uncontrolled process becomes temporarily controlled. See the manual page for KINIT() for further discussion of this topic.

| | |
|---|---|
| **IGNDID()** | Get the local nodeid. |
| **IGNDTP()** | Get the local node type. |
| **IGRTP()** | Get the type of a remote node. |
| **IGORGN()** | Get the origin OTB nodeid. |
| **IGNITB()** | Get total number of ITB nodes. |
| **IGNOTB()** | Get total number of OTB nodes (always 1). |
| **IGNALL()** | Get total number of nodes. |
| **IGNCST()** | Get total number of castids at local node. |
| **GNET()** | Get comprehensive network information. |
| **IGNTP()** | Get total number of nodes of specific type. |
| | |
| **ORIGIN** | IGORGN() |
| **LOCAL** | special identifier for the local node |

## Network and Node Information

It is often necessary, especially for message passing, to identify specific nodes in the booted system. Identifiers for the nodes are established at boot time, based on information stored in a special file called a boot schema. Node identifiers may vary from schema to schema. The most commonly used nodeids are the local nodeid and the origin OTB nodeid. These two values, in addition to the local node type, are returned by the GNET() subroutine after obtaining them from the local route daemon.

```
integer function IGNDID()
integer function IGORGN()
integer function IGNDTP()
```

```
subroutine GNET(nodeid, nodetype, origin, ierr)
integer nodeid, nodetype, origin, ierr
```

The first three functions call GNET(), cache the information and return the appropriate value returned argument. Subsequent calls simply return the value from the cache. The special constant nodeid LOCAL can be used to indicate the local node without knowing its actual nodeid. The macro ORIGIN is always replaced by IGORGN(). These are defined in <NET.h>.

### Node Types

The node type is a bit mask containing one or more attributes. The attributes are defined in <NET.h> and are listed below.

| | |
|---|---|
| **NT_OTB** | OTB node |
| **NT_ITB** | ITB node |
| **NT_CAST** | multicast node |

| | |
|---|---|
| **NT_WASTE** | node not available to user |
| **NT_DISK** | node with attached disk |
| **NT_TUBE** | node with attached video |
| **NT_JONES** | neighbour node |
| **NT_BOOT** | child node in boot tree |

The type of any node is returned by the IGRTP() function.

```
integer function IGRTP(nodeid)
integer nodeid
```

It is often useful to count the number of nodes matching certain node type attributes. The IGNTP() function requires the caller to give a node type and a node type mask indicating which bits in the node type are to be considered[1]. Other functions of the form IGN*() are simply wrappers for IGNTP(). IGNCST() counts all cast identifiers whose spanning tree passes through the local node (see *Multicasting and Multireeling*).

```
integer function IGNTP(type, mask)
integer type, mask
integer function IGNITB()
integer function IGNOTB()
integer function IGNALL()
integer function IGNCST()
```

1. IGNTP(type, mask) counts a node type, nt if (AND(nt, mask) .eq. AND(type, mask)).

| | |
|---|---|
| **KSEND()** | Send a local message. |
| **KRECV()** | Receive a local message. |
| **KTSEND()** | Try to send a local message. |
| **KTRECV()** | Try to receive a local message. |

## Kernel Message Passing

There are two levels of message passing in Trollius. The kernel, or local level allows communication between processes on the same node. The network level allows communication between processes on different nodes, as well as between processes on the same node. In fact, the network level is constructed solely with kernel level message passing. This section deals with the kernel level.

A process sending a message does not directly specify the pid of the receiving process. Instead, each process specifies an arbitrary 32 bit quantity called an event and an arbitrary 32 bit quantity called a message type. As an implicit, hidden part of the message passing procedure, each process sends a message (using the underlying environment) to the kernel. If the events and types "match", the kernel sends reply messages to both parties. The reply message contains the information necessary for the sending process to send a message to the receiving process directly using the underlying environment. From the user's point of view, two processes synchronize and a message is passed.

The kernel offers a non-blocking variation of the send and receive functions. Blocking functions do not return until synchronization and data transfer has occurred. Non-blocking functions return an error if synchronization cannot take place immediately. From these rules we can deduce that on the local node where only one process can run at any one time, two processes will never synchronize if they both use non-blocking functions. A non-blocking function that did not synchronize returns EWOULDBLOCK.

The blocking functions are KSEND() and KRECV(). The non-blocking functions are KTSEND() and KTRECV().

```
subroutine KSEND(kevent, ktype, klength, kflags,
     kmsg, ierr)
subroutine KRECV(kevent, ktype, klength, kflags,
     kmsg, ierr)
```

```
subroutine KTSEND(kevent, ktype, klength, kflags,
     kmsg, ierr)
subroutine KTRECV(kevent, ktype, klength, kflags,
     kmsg, ierr)

integer kevent, ktype, klength, kflags, ierr
integer kmsg(*)
```

All arguments must be explicitly set prior to calling the subroutine. Some arguments will be changed when the function returns.

The kevent argument holds the event, a positive integer chosen by the user. The events specified by the sending and receiving processes must be equal. Negative events are reserved for system processes.

The ktype argument is also used to match sending and receiving processes. This is an integer, used as a array of bits. If the type argument is set to 0, the message can synchronize with any process matched on the event. If it is non-zero, a matching process must have at least one bit set in a corresponding position in its ktype argument. In other words, the bitwise logical AND (*&*) of the two type arguments must not be 0. This argument is set to the sender's type after successfully returning from KRECV() or KTRECV().

The klength argument specifies the length, in bytes, of the message to be sent, or the size of the receiving buffer. If the sender and receiver specify different lengths, the smaller amount will be transferred. This argument is set to the amount transferred after calling any message passing subroutine that returns successfully (synchronizes without error).

The kflags argument is normally set to 0. It can be set to KHOLD (defined in <KREQ.h>) to establish a virtual circuit (see *Kernel Virtual Circuits*).

The kmsg argument is set to the address of the buffer holding the data to be sent, or the address into which the data is to be received.

## Kernel Virtual Circuits

Two processes, using KSEND() and KRECV(), can establish a circuit that will remain in effect for the transmission of several messages. These messages (except for the first one) will synchronize directly with the process at the other end of the circuit, ignoring the event and type and bypassing the kernel, thus decreasing the overhead of message passing.

This is accomplished by setting the kflags argument to KHOLD in both the sending and receiving process. The virtual circuit thus established is associated with the event[1] and is used for the passing of the **next** message with that event. The virtual circuit remains open until a message is passed on that event with the KHOLD bit of both kflags arguments reset to 0. While open, the KSEND() and KRECV() subroutines will use the virtual circuit in a manner transparent to the user. Virtual circuits do not support non-blocking synchronization.

A virtual circuit is bidirectional, and there can be up to 16 simultaneous virtual circuits for each process. The process can continue to send messages on non-circuit events; it will synchronize through the kernel for these events as usual. Events associated with virtual circuits must differ in the last four bits.

After the first message establishing the virtual circuit, there is no protection against length mismatch. The underlying environment may cause the processes to fail (crash or return an error) if the lengths differ. Above all, virtual circuits should always be used with caution.

---

1. The type is not used in identifying an existing virtual circuit. It is used, as normal, in synchronizing the establishment of a new virtual circuit.

| | |
|---|---|
| **TSEND()** | Send transport message. |
| **TRECV()** | Receive transport message. |
| **NSEND()** | Send network message. |
| **NRECV()** | Receive network message. |
| **NTSEND()** | Send non-blocking network message. |
| **NTRECV()** | Receive non-blocking network message. |
| **DSEND()** | Send datalink message. |
| **DRECV()** | Receive datalink message. |
| **DTSEND()** | Send non-blocking datalink message. |
| **DTRECV()** | Receive non-blocking datalink message. |
| **PSEND()** | Send physical message. |
| **PRECV()** | Receive physical message |

## Network Message Passing

There are four sub-levels of message passing within the Trollius network level, representing different functionality/overhead compromises. They are, in order of increasing functionality and overhead, the physical, datalink, network and transport sub-levels. The primary feature of all these sub-levels is the ability to send a message to a remote (non-local) node. The ITB physical sub-level utilizes hardware links directly and efficiently. The datalink sub-level offers multiple process arbitration of a link, but has no built-in routing capabilities. The network sub-level adds routing and large message packetization to the functionality of the datalink sub-level. The transport sub-level manages message flow control; a message will not be transmitted until the receiving process is ready. The network sub-level functions are the most commonly used, most generic communication functions in Trollius.

The arguments passed to all the network level subroutines are an extension of the kernel level arguments used by KSEND() and KRECV(). Some arguments will be altered upon return of a subroutine.

Network message passing can be as simple or powerful and complex as the user desires. By sticking to the network sub-level and ignoring half the fields in the network message descriptor, the user will be dumb, happy and productive. Performance and fine control require more understanding of the operating system and even the hardware.

The network subroutine arguments are used generally in the same way by each sub-level. We will discuss the general use here and then point out peculiar usage during a discussion of each sub-level.

| | |
|---|---|
| **`ndlevent`** | This argument is used by the sending process at the datalink and physical sub-levels to indicate the event of the forwarding process. Usually, this is the event of a datalink output process, the proprietor of a communication link. |
| **`nnode`** | The sending process places the destination nodeid into this argument. The receiving process does not use this field. The receiving process cannot specify the sending node. Node filtering must be accomplished with the nevent and/or ntype arguments. This is very important. |
| **`nevent`** | This is the synchronization event at the destination node. The usage is exactly the same as the kernel level. |
| **`ntype`** | This is the synchronization type at the destination node. The usage is exactly the same as the kernel level. |
| **`nlength`** | The rules for message length and minimum transfer are the same as the kernel level. The difference is that the sender in not informed of the length received. |
| **`nflags`** | This is a bit field of options, similar to the kernel level. Indeed, the kernel level KHOLD flag can be used. Flags are defined in <NET.h> and <KREQ.h>. |
| **`ndata`** | The network level provides a small pouch, referenced by this argument, that may contain data not stored contiguously in memory with the main message. |
| **`ndsize`** | This argument gives the length of the data pouch, whose factory default maximum is eight words. |
| **`nmsg`** | The usage is exactly the same as the kernel level. |

The next subsections will discuss each network message passing sub-level, in order of decreasing functionality and overhead. This is also the order that would be employed in successive optimizations of an appropriate application program.

**Transport Sub-level**

The transport sub-level subroutines TSEND() and TRECV() ensure process synchronization and flow control. A transport sub-level message will not be sent until the intended receiver has called TRECV(). This prevents the sender from swamping the message buffers, or jamming the communication links, with a large number of messages. Synchronization is provided at the price of lower performance. Non-blocking subroutines are not available at the transport sub-level.

```
subroutine TSEND(nnode, nevent, ntype, nlength,
     nflags, ndata, ndsize, nmsg, ierr)
subroutine TRECV(nevent, ntype, nlength,
     nflags, ndata, ndsize, nmsg, ierr)
```

```
integer nnode, nevent, ntype, nlength, nflags
integer ndata(*), ndsize, nmsg(*), ierr
```

`nflags` NOBUF, BYBUF, DRAWDATA, DINTDATA, DFLTDATA, DDBLDATA, DRAWMSG, DINTMSG, DDBLMSG

The best application of the transport sub-level is debugging communication. Virtual circuits (see *Network Virtual Circuits*) are not available at this sub-level.

**Network Sub-level**

The network sub-level is the most common form of Trollius message passing. It does not have the flow control of the transport sub-level, but it has everything else. The chief features are routing and long message packetization.

The network sub-level subroutines are NSEND() and NRECV(). NSEND() can send a message to any node in the multicomputer. It uses the nnode argument to determine which of the local node's communication links to use in transmitting the message. If nnode matches the local nodeid or nnode holds the special nodeid, LOCAL, the message is sent to the local process matching the message event and type. This will happen eventually when the message arrives at the destination node. The NRECV() subroutine does not distinguish between the source nodes of multiple incoming messages. The nnode field is unused by NRECV() and the first message that matches the event and type is received.

The maximum length of a network message is defined by MAXNMSGLEN in <NET.h> and is usually set at 4096 bytes. Longer messages are broken up into packets that respect this upper bound. The decomposition takes place inside NSEND() and the recomposition is done by NRECV(). All the packets have the same node specified in the network message descriptor. Apart from the network sub-level runtime functions, Trollius is very ignorant of packetization.

```
subroutine NSEND(nnode, nevent, ntype, nlength,
     nflags, ndata(*), ndsize, nmsg(*), ierr)
subroutine NRECV(nevent, ntype, nlength, nflags,
     ndata(*), ndsize, nmsg(*), ierr)
```

```
integer nnode, nevent, ntype, nlength, nflags
integer ndata(*), ndsize, nmsg(*), ierr
```

NTSEND() and NTRECV() are called with the same arguments as NSEND() and NRECV(), respectively.

**`nflags`** KHOLD, NOBUF, BYBUF, NREEL, DRAWDATA, DINTDATA, DFLTDATA, DDBLDATA, DRAWMSG, DINTMSG, DDBLMSG

The non-blocking NTRECV() subroutine synchronizes very much like the KTRECV() kernel level subroutine. However, the NTSEND() subroutine may behave quite differently due to the presence of message buffers between the sending and receiving processes. NTSEND() can be successful even if the receiving process has not called NRECV() because the measure of success is only the ability to get rid of the message. If a free buffer exists, it stores the message and NTSEND() returns a 0 in the ierr argument.

The network sub-level is used by most applications and by most Trollius system programs. No program should attempt the use of the lower sub-levels without first demonstrating correct operation with this sub-level. If the overhead of this sub-level is acceptable, the user need not look any further.

**Datalink Sub-level**

Successful parallel applications rely on localized communication for performance and scalability. Much communication may be restricted to the immediate (nearest) neighbours of a node. If the packetization service of the network sub-level is not needed, the datalink functions DSEND() and DRECV() may be used for nearest neighbour message passing.

Much of the general functionality of the network level is provided by the datalink sub-level. In particular, we will discuss buffering and virtual circuits as part of the datalink sub-level, although they may also be accessed from the network and, in the case of buffering, the transport sub-levels.

The datalink sub-level functions are DSEND() and DRECV(). DRECV() is used in exactly the same way as NRECV(). To use DSEND(), the sending process explicitly names the output link on the local node in the ndlevent argument. There are two ways to obtain the correct link information.

- A link number, in the range 0 to L - 1, where L is the number of links on a node is converted by the function IDLEVT() to the information needed by DSEND().

- The nodeid of the neighbour node is converted by the subroutine, GROUTE(), to the information needed by DSEND(). The extra event argument is returned by GROUTE() when the nodeid argument is the local nodeid. This enables DSEND() to send a message to a receiving process on the local node.

```
integer function IDLEVT(link)
subroutine GROUTE(nodeid, event, ierr)
integer link, nodeid, event, ierr
```

```
subroutine DSEND(ndlevent, nevent, ntype, nlength,
     nflags, ndata, ndsize, nmsg, ierr)
subroutine DRECV(nevent, ntype, nlength, nflags,
     ndata, ndsize, nmsg, ierr)
integer ndlevent, nevent, ntype, nlength, nflags
integer ndata(*), ndsize, nmsg(*), ierr
```

DTSEND() and DTRECV() are called with the same arguments as DSEND() and DRECV(), respectively.

**ndlevent** This argument is used the sending process calling DSEND(). It holds a value associated with the output link connecting the destination neighbour node. It can also hold the event of a local receiving process. This argument is not present in DRECV().

**nflags** KHOLD, NOBUF, BYBUF, DRAWDATA, DINTDATA, DFLTDATA, DDBLDATA, DRAWMSG, DINTMSG, DDBLMSG

**nlength** There is no packetization at the datalink sub-level. Message length is limited to MAXNMSGLEN.

**Network Message Buffers**

Message buffering is a separate, modular feature of the network level directly accessed by the datalink sub-level but also accessible to the transport and network sub-levels. Without special option flags in the nflags argument of the network subroutines, buffers will be used when available and needed. Two option flags in nflags limit a message's buffer use.

**NOBUF** The message will never use buffers.

**BYBUF** The message will not use buffers on the local node.

If these options are left out of nflags, the message may use a buffer anywhere along the route to the destination node, though it is not required. Passing through a buffer constitutes a large overhead for message passing. If an application can guarantee that buffers will not be needed, the NOBUF option is recommended Only the sending subroutines use these options.

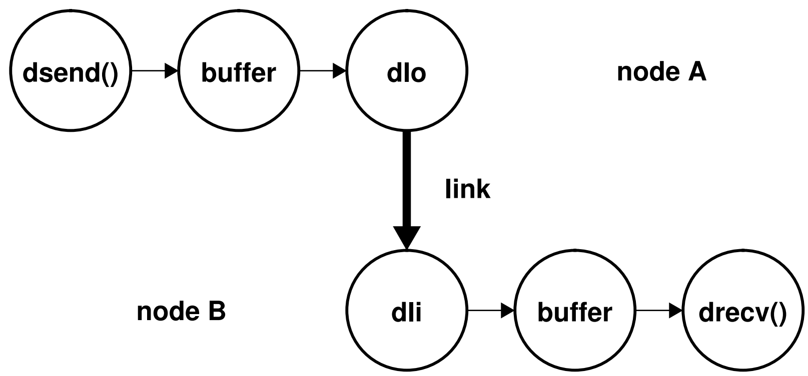


**Figure 2: Network Message Buffering**

The datalink sub-level message transfer, depicted in Figure 2, can involve buffers on the sending and receiving nodes. The BYBUF option locks out the first buffer opportunity in the DSEND() to DRECV() path but not the second. The NOBUF option locks out both buffer opportunities. Figure 2 also shows the special Trollius system processes that own and operate communication links between nodes. They are called datalink processes and dlo is shorthand for datalink output; dli is shorthand for datalink input. The importance of these processes to the application programmer is that they each may hold one buffered message.[1]

Let us take a closer look at the buffer process. There are really two types of processes at work. The buffer daemon collects all messages. The buffer daemon forwards messages to a limited number of buffer workers. Messages that cannot immediately be forwarded to a buffer worker are kept inside the buffer daemons internal pool of message buffers. Each buffer worker can

1. An exception to this rule is the optimized datalink output process that runs on most ITB transputer links (dlo_t4). This process cannot hold a buffered message.

only buffer one message. The buffer worker sends its message to the original destination, as if the buffer detour had not been made.

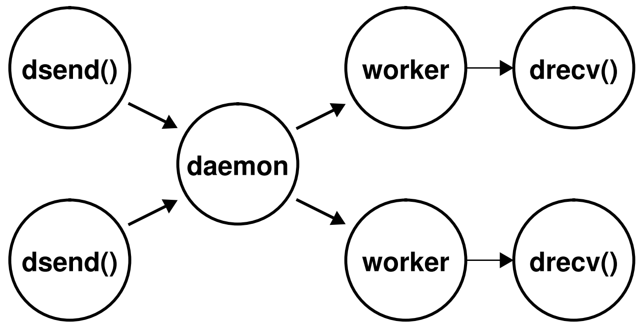


**Figure 3: Buffer Daemon and Buffer Worker**

Figure 3 illustrates the buffer daemon/worker structure. Notice that if a message passes through a buffer, it must pass through a worker to reach the receiving process. If the workers are full and cannot proceed, the message is stuck in the daemon's pool and the receiving process is permanently blocked. This is a deadlock.

It is the buffer daemon's policy to not have two workers busy with two messages of identical event and message type. This sets up a potential hazard for receiving processes that use several events and/or several types with restrictive matching. Such processes are effectively state machines. They synchronize on one event/type and then, perhaps dependent on the input data, synchronize on another event/type. This is a bad communication structure for the distributed memory architecture, but it is commonly used by application programmers. What happens if the local node's memory is filled by messages the receiver won't receive until another message event/type has arrived? The other message can't arrive because the memory is full. Trollius network message buffering speeds up this problem by replacing the local memory with buffer workers as the critical resource[1]. A safe receiving process can receive any of its message classes at any time. A really safe receiving process buffers its messages internally. You can rely on Trollius buffers

1. The reward for this buffer process structure is that network message buffering remains outside the kernel and very flexible at runtime. The kernel remains ignorant of the network.

if your receiving process switches between only a few unique event/type combinations. Avoid using large numbers of event/type combinations or a deadlock will be your reward.

**Network Virtual Circuits**

Network virtual circuits are a feature of the datalink sub-level that is also available to the network and transport sub-levels. A circuit can be created between any two processes in the multicomputer. The most common use is at the datalink sub-level between processes residing on neighbour nodes. The circuits offer substantially reduced overhead over several messages at the expense of multiple process access to the communication link. When a sending process creates a network virtual circuit to a neighbour node, the connecting communication output link is effectively reserved to the sending process. No other process can use the output link, including system processes. A receiving process reserves the communication input link in the same way. A circuit that goes beyond the neighbour reserves all the links along the route. Multiple simultaneous and not carefully planned remote circuits can quickly lead to deadlock. Concentrate on using neighbour circuits, but be careful to turn them off before the process exits, to allow the system to resume using the links.

The rules for network virtual circuit creation and destruction are the same as for the kernel level (see *Kernel Virtual Circuits*). They are activated by the same option flag, KHOLD. Unlike the kernel level, network virtual circuits are unidirectional. Another difference is that network virtual circuits check and handle sender/receiver length mismatch.

In this subsection we present an example of datalink sub-level virtual circuit programming. The send and receive programs are close reflections of each other.

```
C
C Send 10 messages out link 1 using a virtual
C circuit.
C
#include <NET.h>
     program send
     integer*4      buffer(20)
C
C Attach to Trollius kernel.
C
     call KINIT(0, ierr)
```

```
      if (ierr .ne. 0) then
           call KEXIT(ierr)
      endif
C
C The first message opens the virtual circuit.
C We are separating this message from the
C body messages for easy conversion to the
C physical sub-level.
C
      ndlevent = IDLEVT(1)
      call DSEND(ndlevent, 11, 0, 80, KHOLD, 0, 0,
           buffer, ierr)
      if (ierr .ne. 0) then
           call KEXIT(ierr)
      endif
C
C Send the body messages.
C
      do 10 i = 1,8
           call DSEND(ndlevent, 11, 0, 80, KHOLD,
                     0, 0, buffer, ierr)
           if (ierr .ne. 0) then
                     call KEXIT(ierr)
           endif
10    continue
C
C The last message closes the virtual circuit.
C
      call DSEND(ndlevent, 11, 0, 80, 0, 0, 0,
           buffer, ierr)
      if (ierr .ne. 0) then
           call KEXIT(ierr)
      endif
C
C Detach from Trollius kernel.
C
      call KEXIT(0)
      stop
      end
```

The receive program must match the send program exactly on KHOLD synchronization. A mistake here will cause the program to hang.

```
C
C Receive 10 messages using a virtual circuit.
C The receiver does not specify the input link.
C
#include <NET.h>
      program recv
      integer*4      buffer(20)
C
C Attach to Trollius kernel.
C
      call KINIT(0, ierr)
      if (ierr .ne. 0) then
           call KEXIT(ierr)
      endif
C
C The first message opens the virtual circuit.
C We are separating this message from the
C body messages for easy conversion to the
C physical sub-level.
C
      call DRECV(11, 0, 80, KHOLD, 0, 0,
           buffer, ierr)
      if (ierr .ne. 0) then
           call KEXIT(ierr)
      endif
C
C Receive the body messages.
C
      do 10 i = 1,8
           call DRECV(11, 0, 80, KHOLD, 0, 0,
                     buffer, ierr)
           if (ierr .ne. 0) then
                     call KEXIT(ierr)
           endif
10    continue
```

```
C
C The last message closes the virtual circuit.
C
      call DRECV(11, 0, 80, 0, 0, 0,
                    buffer, ierr)
      if (ierr .ne. 0) then
           call KEXIT(ierr)
      endif
C
C Detach from Trollius kernel.
C
      call KEXIT(0)
      stop
      end
```

**Data Representation and Conversion**

Message data, excepting text, must be converted to the proper machine representation and byte order when transmitted between nodes with different processors. The datalink sub-level can make the necessary conversions when a message crosses the link between differing nodes. The typical multicomputer may have a need for conversion between the OTB node and the ITB nodes. Data representation conversion is also available at the network and transport sub-levels.

By default, the data pouch (ndata) argument is assumed to hold integer data, and the data representation is converted to the appropriate integer byte ordering during network message passing. The message body (nmsg) is not altered by default. The default behavior can be changed with option flags in the nflags argument of the sender's network subroutine call. The following flags, defined in <NET.h>, each assume a data type and make the appropriate change in data representation. They will have no effect if no data conversion is needed.

| | |
|---|---|
| **DINTDATA** | ndata holds integers |
| **DFLTDATA** | ndata holds floats |
| **DDBLDATA** | ndata holds doubles |
| **DRAWDATA** | ndata representation will not be changed |
| **DINTMSG** | nmsg holds integers |
| **DFLTMSG** | nmsg holds floats |
| **DDBLMSG** | nmsg holds doubles |
| **DRAWMSG** | nmsg representation will not be changed |

If the data pouch or message body contain data of mixed types, you will need to convert each type separately. Trollius supplies functions that convert data between the local representation (regardless of what it is) and a standard (Trollius) representation. Data should be converted to Trollius representation before sending, and converted to local representation after receiving. You will need to use the functions appropriate to the given data type.

| | |
|---|---|
| **ILTOT()** | converts integer*4 representation from local to Trollius |
| **ITTOL()** | converts integer*4 representation from Trollius to local |
| **ILTOTS()** | converts integer*2 representation from local to Trollius |
| **ITTOLS()** | converts integer*2 representation from Trollius to local |
| **FLTOTF()** | converts real*4 representation from local to Trollius |
| **FTTOLF()** | converts real*4 representation from Trollius to local |
| **ILTOTD()** | converts double representation from local to Trollius |
| **ITTOLD()** | converts double representation from Trollius to local |

The syntax for all the above functions is of the form:

```
cvar = ILTOT(var)
```

where cvar and var are of the appropriate data type for the given function. Var is the original value and cvar is the value converted to the appropriate data representation.

**Physical Sub-level**

The physical sub-level subroutines PSEND() and PRECV() directly access hardware communication links. They may be used between ITB node neighbours only, and only after a network virtual circuit has been established between the two nodes (see *Network Virtual Circuits*). When physical sub-level communication is completed, the circuit must be terminated at the datalink or network sub-level.

```
subroutine PSEND(ndlevent, nlength, ndata, ndsize,
     nmsg, ierr)
subroutine PRECV(nevent, nlength, ndata, ndsize,
     nmsg, ierr)
integer ndlevent, nlength, ndata(*), ndsize
integer nmsg(*), nevent, ierr
```

**ndlevent** As with the datalink sub-level, the sending process uses this argument to identify the output link.

**nevent** The receiving process maintains this argument with the value used to synchronize the virtual circuit.

**nlength** There is no protection against sender/receiver length mismatch at the physical sub-level.

Unlike the other network level subroutines, PSEND() and PRECV() do not handle the truncation that may be needed when the sending process specifies a length different than that specified by the receiving process. If the processes do not specify equal lengths, they will likely hang.

**Basic Network Communication**

A special pair of subroutines provide a simplified interface to the network sub-level. BSEND() and BRECV() are most appropriate for the novice Trollius programmer. These “basic” subroutines hide the event and the type synchronization points, leaving the programmer to concentrate on nodeids only. Only one process per node using BSEND() and BRECV() is allowed.

```
subroutine BSEND(nnode, nmsg, nlength,
     nflags, ierr)
subroutine BRECV(nnode, nmsg, nlength,
     nflags, ierr)
integer nnode, nmsg(*), nlength, nflags, ierr
```

**nnode** This is the destination node for BSEND() and the source node for BRECV().

**nflags** The conversion flags indicate the data element type.

**nlength** This is expressed as the number of data elements.

| | |
|---|---|
| **RCAST()** | Register a castid. |
| **RMELT()** | Remove a castid. |
| **RCASK()** | Get number of branches at given node. |
| **RRNDWT()** | Wait for a castid to be defined. |

## Multicasting and Multireeling

Multicasting allows a message to be sent to a group of nodes, using a single call to the network sub-level function, NSEND(). The reverse procedure, multireeling, receives a message from a group of nodes. The group of nodes representing the multicast destination must be registered under a cast identifier (castid). The castid can then be used instead of a nodeid in the nnode argument of NSEND().

The RCAST() function handles castid registration.

```
subroutine RCAST(nodeid, castid, combine,
     pcast, ncast)
integer nodeid, castid, combine, pcast(*), ncast
```

A castid defines not only the destination nodes, but also the source node. The first argument defines the source node. Though the registration can be initiated from any node, it is usually done from the multicast source node and the first argument may be given as LOCAL (defined in <NET.h>).

A castid exists within the domain of nodeids and must be unique among all nodeids in the Trollius multicomputer and all other castids. With this restriction, the user may choose any unused identifier.

The pcast argument is an array of nodeids of length ncast. These are the destination nodes of the castid.

During registration, the system defines a spanning tree across the node network, beginning at the source node and reaching every destination node. The system remembers this spanning tree and uses it every time a message is sent to the castid from the source node. At each fork in the tree, the message is copied and sent to all branches.

Multireeling is more complicated. The leaves of the spanning tree become source nodes send a message to the castid. They must set the NREEL option in the nflags argument of NSEND() to distinguish a multireel from a multicast. As messages meet at forks in the spanning tree, they are combined into one message and sent on towards the top of the tree, now the destination node. The multireel messages are combined pair-wise until all messages arrive at a fork node and are combined. (The fork node itself may be a destination node.) The method by which two messages are combined into one message is defined by the user. The system supplies several simple built-in combining methods which are associated with a castid during the RCAST() subroutine. The combine argument to RCAST() is set to one of the values listed below and defined in <NET.h>.

| | |
|---|---|
| **CBN_ERR** | combine error codes in first integer only |
| **CBN_ADD** | combine by adding all corresponding integers |
| **CBN_MIN** | combine by taking the minimum |
| **CBN_MAX** | combine by taking the maximum |
| **CBN_AND** | combine by applying a bitwise AND |
| **CBN_OR** | combine by applying a bitwise OR |
| **CBN_XOR** | combine by applying a bitwise XOR |

A separate process can combine multireeled messages. In this case the combine argument to RCAST() is set to an event used by the combining process. Combining can be done by a single application process, given the number of branches entering the local node. This important value is returned by the RCASK() subroutine. For a detailed discussion of multireel combining strategies, see the manual page for RCAST().

```
subroutine RCASK(nodeid, castid, nbranch, ierr)
subroutine RMELT(nodeid, castid, ierr)
subroutine RRNDWT(nodeid, castid, ierr)
```

A multireel should not be attempted before the castid is defined. The RRNDWT() subroutine can be used by the multireel source nodes to ensure that a castid is defined. When a castid is no longer needed, the spanning tree information can be freed with the RMELT() subroutine.

**Caveats**

The 2.2 multireel service has caveats.

- Avoid multiple simultaneous multireels on the same castid and the same event.
- Reeled messages must all be the same length.
- The data pouch in reeled messages is limited to six words.

| | |
|---|---|
| **`TOPEN()`** | Open a file. |
| **`TCLOSE()`** | Close a file. |
| **`TREAD()`** | Read from a file. |
| **`TWRITE()`** | Write to a file. |
| **`IUTOD()`** | Learn descriptor behind unit. |
| **`TSINGL()`** | Switch to single access method. |
| **`TMULTI()`** | Switch to multiple access method. |
| **`TASYNC()`** | Switch to async access method. |
| **`RFORDR()`** | Change multiple node ordering. |

## File Access

The OTB node in a Trollius multicomputer allows remote access to its file system via a standard UNIX interface. ITB nodes with a disk and a file system may also provide remote access in the same way. Trollius does not provide a file system, only remote access to a file system from any node.

The actual remote UNIX file functions, which access the Trollius operating system, are written in C. Only a few key functions are wrapped for Fortran programmers. The classic method of Fortran I/O is via the intrinsic unit I/O functions. Neither level of I/O will be fully described here, but note that Trollius specific file access features are available only to the lower level.

File pathnames refer to files on the OTB node's operating system by default. However, a specific nodeid can be attached to a pathname with the following syntax:

**`nodeid:path`**

Each Trollius process may have a maximum of 32 simultaneously open file descriptors. Remember that all Trollius UNIX file functions involve message passing using the same links, buffers and other resources used by an application.

Some Trollius specific features of remote file access are controlled by additional flags in the flags argument of the TOPEN() subroutine. These flags, defined in <FREQ.h> are listed below.

| | |
|---|---|
| **`O_LOCK`** | Lock the file descriptor into the remote file server's open descriptor cache. See the manual page rfunix(). |
| **`O_SINGL`** | Open the file using the single access method. See *Parallel File Access*. |

| | |
|---|---|
| **`O_MULTI`** | Open the file using the multiple access method. See *Parallel File Access*. |
| **`O_REUSE`** | Reuse existing open file descriptor for matching pathname and open flags if found. This is useful for asynchronous access to one open file with one file pointer. |
| **`O_1WAY`** | Write to the file without waiting for completion or return code. This greatly increases write performance but should only be used on a debugged application. |

**Parallel File Access**

Trollius supports a loosely synchronous, parallel I/O access method based on Cal Tech's Cubix. One process on every ITB node must participate in Cubix access method functions. A file operation is not completed until all client processes have called the same file subroutine. There are two different Cubix access methods that solve two common file read/write problems in data parallel programming.

| | |
|---|---|
| single | All clients execute the same file subroutine with the same amount of identical data. The data from only one (arbitrary) client is transferred. This is useful when all clients want to read a global value from a file, or write a global value to a file. It is especially convenient during I/O to a terminal. The user enters data once and it is read by all nodes. All nodes print a prompt or an error message and it appears once on the terminal. |
| multiple | All clients execute the same file subroutine with different amounts of different data. All the data from all the clients is transferred, but the order of transfer is strictly controlled. By default, the lowest value nodeid will transfer first and the sequence continues until the highest value nodeid transfers last. This is useful in decomposing a data structure during read so that the right nodes get the right subset of data and in recomposing a data structure during write so that the data structure is not jumbled. |

Without Cubix file access, an application often needs a controlling program to manage the parallel processes and filter I/O. Cubix can eliminate the need for a control program. The non-Cubix default access method is also called the asynchronous method (async). Without synchronization, a message written by N nodes appears N times on the terminal. A decomposed data structure written to a file appears in a random order.

The RFORDR() function changes the default order of client data transfer in the Cubix multiple method. Each client specifies a unique sequence number from 0 to N-1, where N is the number of clients (ITB nodes)[1].

```
subroutine RFORDR(fd, sequence, ierr)
integer fd, sequence, ierr
```

A file is opened with a Cubix access method by using one of the special open flags, O_SINGL or O_MULTI. The access method of a file can be changed with the functions listed below. In each case, one client process on each ITB node must call the same subroutine. When going from asynchronous to a Cubix access method, all async file descriptors must match in pathname and non-Cubix open flags. More commonly, applications will need to switch a file between single and multiple access methods. See the manual page for cubix for more information.

```
TSINGL(fd, ierr)
TMULTI(fd, ierr)
TASYNC(fd, ierr)
integer fd, ierr
```

**Cubix Example**

We present a simple example program that reads the total size of a one dimensional array from a file, then reads its subset of the array based on the total number of ITB nodes.

```
C
C Read and decompose a 1-D array of reals
C across a 1-D array of ITB nodes.
C Assume the array length decomposes evenly
C across the number of ITB nodes.
C
#include <FREQ.h>
     program        cubix
     real*4         data(100)
     integer        fd
     integer        global_len
     integer        local_len
     integer        nread
```

1. Cubix places no restriction on the assignment of nodeids in the boot schema.

```
C
C Open the file first with Cubix singl method.
C This is not an error handling tutorial.
C
      call TOPEN(fd, "data", O_RDONLY + O_SINGL,
           0, ierr)
C
C Read the global (total) length of the array.
C
      call TREAD (fd, global_len, 4, ierr)
C
C Switch to Cubix multiple method.
C
      call TMULTI(fd, ierr)
C
C Calculate the local length and read the local
C subset of the data.
C
      local_len = global_len / IGNITB()
      call TREAD(fd, data, local_len * 4,
           nread, ierr)
C
C Close the file.
C
      call TCLOSE(fd, ierr)
      stop
      end
```

| | |
|---|---|
| **RTVCLR()** | Clear graphics frame. |
| **RTVRCT()** | Plot rectangle of pixels. |
| **RTVSNK()** | Plot snake of pixels. |
| **RTVTXT()** | Plot text on graphics frame. |
| **RTVSTA()** | Get status of graphics frame. |

## Data Visualization

Trollius provides a simple interface to the graphical capabilities of a node, provided they exist. Before these functions can be used intelligently, some basic information must be learned about the graphics frame buffer. The RTVSTA() subroutine has several value returned arguments.

```
subroutine RTVSTA(nodeid, xlen, ylen, ncol,
     xtext, ytext, ierr)
integer nodeid, xlen, ylen, ncol, xtext
integer ytext, ierr
```

| | |
|---|---|
| **xlen** | width of frame buffer in pixels |
| **ylen** | height of frame buffer in pixels |
| **ncolour** | length of colour palette |
| **xtext** | width of frame buffer in characters |
| **ytext** | height of frame buffer in characters |

The RTVRCT() subroutine plots a rectangle of pixels anywhere inside the frame buffer. The upper left pixel coordinates and the pixel dimensions of the rectangle are given. Each pixel is a byte (0 - 255) index into the colour map.

```
subroutine RTVRCT(nodeid, x1, y1, xlen, ylen,
     colours, ierr)
integer nodeid, x1, y1, xlen, ylen, ierr
character colours(*)
```

The RTVSNK() subroutine is similar to RTVRCT(). A single starting pixel ordinate is given and the given number of pixels are plotted along the horizontal rows of the frame buffer.

```
subroutine RTVSNK(nodeid, x1, len, colours, ierr)
integer nodeid, x1, len, colours, ierr
character colours(*)
```

The RTVTXT() subroutine views the screen in characters, not pixels, but otherwise operates much like RTVSNK(). At starting character coordinates

RTVTXT() plots the null terminated ascii character array across the horizontal character rows of the frame buffer. One colour is given for all characters.

```
subroutine RTVTXT(nodeid, x1, y1, colour,
     text, ierr)
integer nodeid, x1, y1, colour, ierr
character text(*)
```

The RTVCLR() subroutine sets all pixels in the frame buffer to the background colour (black).

```
subroutine RTVCLR(nodeid, ierr)
integer nodeid, ierr
```

| | |
|---|---|
| **`RLEDOP()`** | Operate on remote LEDs. |
| **`RLEDCC()`** | Carbon copy LEDs to remote node. |
| **`RLESTA()`** | Report number of remote LEDs. |

## LED Control and Monitoring

A node may have an attached, programmable LED (Light Emitting Diode) peripheral. Trollius provides a remote service to control these LEDs, mostly for the purposes of demonstration applications, but also for debugging.

A LED is assumed to have two states, on and off. The RLEDOP() subroutine provides several operations that change the state of LEDs. The first argument is the remote nodeid controlling the LEDs. The second argument is the operation code and the third argument is a bit mask, with one bit controlling each LED on a node. A node may have a variable number of LEDs. If a node has N LEDs, the lower order N bits in the bit mask control them.

```
subroutine rledop(nodeid, op, bits, ierr)
integer nodeid, op, bits, ierr
```

The following operations are provided by RLEDOP():

| | |
|---|---|
| **`LEDSET`** | Turn on all LEDs whose bits are set (1) in the bit mask. Do not change the state of LEDs whose bits are reset (0) in the bit mask. |
| **`LEDCLR`** | Turn off all LEDs whose bits are set. |
| **`LEDNOT`** | Toggle all LEDs whose bits are set. |
| **`LEDALL`** | Turn all LEDs whose bits are set. Turn off all LEDs whose bits are reset. |

```
subroutine RLEDCC(nodeid, cc, ierr)
subroutine RLESTA(nodeid, nleds, ierr)
integer nodeid, cc, nleds, ierr
```

RLEDCC() causes one node to reflect the state of all LEDs to a “carbon copy” node. The RLESTA() subroutine returns the number of LEDs on a remote node.

| | |
|---|---|
| **KTRSTR()** | Start tracing and define trace identifier. |
| **KTRSTP()** | Stop tracing. |

## Trace Generation

Traces are records of occurrences in the execution of a Trollius program. Assuming that a process has its trace master switch enabled (done with the command, loadgo; see *Trollius Command Reference*), kernel level functions explicitly control the generation of traces. Also, NSEND() and NRECV() (see *Network Sub-level*) implicitly generate traces.

Besides the master boot-time switch, a program controllable runtime switch also regulates trace generation. Both the master switch and the runtime switch must be on for trace generation to occur. The KTRSTR() and KTRSTP() subroutines each generate a trace, and then turn on or off the runtime switch, respectively.

```
subroutine KTRSTR(tid)
subroutine KTRSTP()
integer tid
```

When a process is created with its master trace switch set in the on position, its runtime switch also begins in the on position. To immediately disable trace generation, KTRSTP() must be called. NSEND() and NRECV() implicitly generate a trace if both switches are on and the synchronization event is positive, a user event.

All traces from the whole Trollius multicomputer are identified in post-processing and visualization steps by a trace identifier (tid). By default, a process generates traces with a tid equal to the nodeid on which the process is running. This is useful for debugging the entire parallel system on the node level. A process is free to change its tid to any value with the KTRSTR() function. It is not necessary to call KTRSTP() first in order to change the tid of a trace enabled process.

| | |
|---|---|
| `TERROR()` | Print an error message. |
| `FTTIME()` | Get value of system timer. |

## Miscellaneous Functions

Every process has the static system error variable, errno. It is set to the appropriate error code upon failure of any Trollius function. Once an error has been detected, a corresponding message can be printed on the standard output with the TERROR() subroutine.

```
subroutine TERROR(message)
character message(*)
```

The caller supplied message is prefixed to the system error message.

Trollius error codes are listed in *Appendix B*.

The function FTTIME() returns the value of the local node's timer in seconds.

```
real*8 function FTTIME()
```

The precision of the timer depends on the underlying environment and/or the hardware of the local node. FTTIME() is usually suitable for timing application performance.

## Appendix A: Constants

The following tables list the values of various system constants used in Trollius 2.2. The user has the option of explicitly coding parameter statements or including the appropriate header files and running the C preprocessor.

| CONSTANT | VALUE | DESCRIPTION |
|---|---|---|
| **LOCAL** | -2147483647 | local node |
| **MAXNMSGLEN** | 4096 | max size of network packet |
| **NHDSIZE** | 8 | max size, in words, of data pouch |
| **NOTEVENT** | -2147483647 | not a valid event |
| **NOTNODEID** | -2147483648 | not a valid node ID |
| **NOTNODETYPE** | -2147483648 | not a valid node type |
| CONSTANT | VALUE | DESCRIPTION |
| **NTOTB** | 0 | Outside The Box (host node) |
| **NTITB** | 1 | Inside The Box |
| **NTCAST** | 2 | multicast of nodes |
| **NTWASTE** | 4 | not fully used |
| **NTDISK** | 8 | node has a disk |
| **NTTUBE** | 16 | node has a video unit |
| **NTJONES** | 32 | neighbour node |
| **NTBOOT** | 64 | child node in boot tree |
| **NTALL** | 31 | match any node type |
| CONSTANT | VALUE | DESCRIPTION |
| **KHOLD** | 1 | set virtual circuit |
| **NOBUF** | 16 | completely bypass buffers |
| **NREEL** | 1024 | multireel |
| **DRAWDATA** | 65536 | don't touch data pouch |
| **DINTDATA** | 131072 | data pouch is integer*4 |
| **DFLTDATA** | 262144 | data pouch is real*4 |
| **DDBLDATA** | 524288 | data pouch is real*8 |
| **DRAWMSG** | 1048576 | don't touch message |
| **DINTMSG** | 2097152 | message is integer*4 |
| **DFLTMSG** | 4194304 | message is real*4 |
| **DDBLMSG** | 8388608 | message is real*8 |
| CONSTANT | VALUE | DESCRIPTION |
| **CBN_ERR** | -1 | error codes in first integer only |
| **CBN_ADD** | -2 | add integers |
| **CBN_MAX** | -3 | maximize integers |
| **CBN_MIN** | -4 | minimize integers |

| | | |
|---|---|---|
| **CBN_XOR** | -5 | exclusive OR integers |
| **CBN_OR** | -6 | OR integers |
| **CBN_AND** | -7 | AND integers |

| CONSTANT | VALUE | DESCRIPTION |
|---|---|---|
| **O_RDONLY** | 0 | read only |
| **O_WRONLY** | 1 | write only |
| **O_RDWR** | 2 | read and write |
| **O_APPEND** | 8 | write append to end of file |
| **O_CREAT** | 512 | create if non-existent |
| **O_TRUNC** | 1024 | truncate to 0 length, then write |
| **O_LOCK** | 65656 | lock in active mode |
| **O_REUSE** | 131072 | use existing descriptor |
| **O_SINGL** | 262144 | use Cubix singular access method |
| **O_MULTI** | 524288 | use Cubix multiple access method |
| **O_NOCBX** | 1048576 | use normal, non-Cubix access method |
| **O_1WAY** | 4194304 | use unidirectional write protocol |

## Appendix B: Error Codes

The Trollius error codes are:

| | | |
|---|---|---|
| **ENOEVDESCRIPTORS** | 201 | out of kernel message descriptors |
| **ECIRCUITINUSE** | 202 | virtual circuit in use |
| **ENOEXDESCRIPTORS** | 203 | out of cleanup function descriptors |
| **EFULL** | 204 | out of descriptors |
| **ENOTPROCESS** | 205 | given process ID does not identify a process attached to the kernel, or no process exists that satisfies the selection parameters |
| **EBADREQUEST** | 206 | invalid request |
| **ESERVICE** | 209 | service unavailable |
| **EMAGIC** | 210 | bad magic number |
| **EBLOWN** | 211 | blown stack |
| **EIMPOSSIBLE** | 212 | impossible, ergo a bug |
| **ENOKERNEL** | 215 | no running kernel |
| **ENOTATTACHED** | 216 | process not attached to kernel |
| **EPANIC** | 217 | lost system integrity |
| **EMSGLEN** | 218 | message length exceeds maximum |
| **EEOF** | 219 | end of file |
| **ECOLLIDE** | 220 | conflict in registering identical multicast identifiers |
| **ESHARE** | 223 | can't find shared segment |
| **EBADLINK** | 224 | invalid link index |
| **EACK** | 225 | bad acknowledgment |
| **EUSAGE** | 226 | command line violates the rules as declared |
| **EOPTION** | 227 | bad/unknown option |
| **ENODERANGE** | 228 | bad node range specification |
| **EBADNODE** | 229 | specified node does not exist |
| **EBADNODESPEC** | 230 | invalid mnemonic node specification |
| **ENORDESCRIPTORS** | 232 | out of route descriptors |
| **ENOBFDESCRIPTORS** | 233 | out of buffer descriptors |
| **ENOBTDESCRIPTORS** | 234 | out of boot descriptors |
| **ENOFLDESCRIPTORS** | 235 | out of flat descriptors |
| **ENOPDESCRIPTORS** | 236 | out of kenya descriptors |

| | | |
|---|---|---|
| **ENOKDESCRIPTORS** | 237 | out of process descriptors |
| **EBADMOSESREQ** | 238 | bad moses request |
| **EMAXBULLETS** | 239 | can't kill datalink |
| **EBADTAG** | 240 | argv, load point, or boot image tag is invalid |
| **EDELETE** | 241 | delete from list failed |
| **ENOBOOTSTRAP** | 242 | no bootstrap present |
| **EBADREC** | 243 | invalid record format |
| **EBADASCIINUMB** | 244 | character string that is supposed to represent a number has invalid format |
| **EHIGHWATER** | 245 | booted system is too big |

The Cubix file error codes are:

| | | |
|---|---|---|
| **ECXREQ** | 260 | badly synchronized Cubix request |
| **ECXMEMBER** | 261 | invalid Cubix member |

UNIX error codes used in Trollius are:

| | | |
|---|---|---|
| **EINTR** | 4 | interrupted system call |
| **E2BIG** | 7 | filename too long |
| **ENOMEM** | 12 | insufficient memory |
| **EINVAL** | 22 | invalid argument |
| **EWOULDBLOCK** | 35 | message sync would block |

**Contact**

Research Computing
The Ohio State University
1224 Kinnear Road
Columbus, OH 43212

**Consulting**

Phone: (614) 292 8492
Internet: trollius@tbag.osc.edu



# Trollius Library Reference in Fortran